ProcessBase Reference Manual

Version 1.0.6

August 1999

Ron Morrison†
Dharini Balasubramaniam†
Mark Greenwood¥
Graham Kirby†
Ken Mayes¥
Dave Munro*
Brian Warboys¥

†School of Mathematical and Computational Sciences,
University of St Andrews

¥Department of Computer Science,
University of Manchester

*Department of Computer Science,
University of Adelaide

Contents

1 Introduction

ProcessBase is the simplest of a family of languages and support systems designed for process modelling. It consists of the language and its persistent environment. The persistent store is populated and, indeed, the system uses objects within the persistent store to support itself. The implication of orthogonal persistence is that the user need never write code to move or convert data for long or short term storage [ABC+83]. The model of persistence in ProcessBase is that of reachability from a root object. The persistent store is stable, that is, it is transformed atomically from one consistent state to the next. Execution against the persistent store is always restarted from the last stable state.

The ProcessBase language is in the algol tradition as were its predecessors S-algol [Mor79], PS-algol [PS88] and Napier88 [MBC+96]. Following the work of Strachey [Str67] and Tennent [Ten77] the languages obey the principles of correspondence, abstraction and type completeness. This makes for languages with few defining rules allowing no exceptions. It is the belief of the designers that such an approach to language design yields more powerful and less complex languages.

The ProcessBase type system philosophy is that types are sets of values from the value space. The type system is mostly statically checkable, a property we wish to retain wherever possible. However, dynamic projection out of unions for type *any* allows the dynamic binding required for orthogonal persistence [ABC+83] and system evolution [MCC+93].

The type system contains the base types integer, real, boolean and string. Higher-order procedures allow code to exist in the value space. Aggregates may be formed using the vector and view types. Both of these allow information hiding without encapsulation. Finally there is an explicit constructor to provide locations.

The type equivalence rule in ProcessBase is by structure and both aliasing and recursive types are allowed in the type algebra.

ProcessBase programs are executed in a strict left to right, top to bottom manner except where the flow of control is altered by one of the language clauses.

The ProcessBase persistent programming system was originally planned as part of the Compliant Systems Architecture Project. It is supported by the EPSRC under grant GR/L32699 at the University of St Andrews and GR/L34433 at the University of Manchester.

The ProcessBase programming system provides the following facilities:

- Orthogonal persistence
  - models of data independent of longevity
- Type completeness
  - no restrictions on constructing types
- Higher-order procedures
  - procedures are data objects
- Information hiding without encapsulation
  - views of data that hide detail
- A strongly typed stable store
  - a populated environment of typed data objects that may be updated atomically

- Hyper-code
  - one representation of a value throughout its lifetime [KCC+92]

- Linguistic reflection
  - to allow reflective programming [Kir92]

- Exceptions
  - for recovering from exceptional conditions

- Interrupts and down-calls
  - for communication between the ProcessBase language and the implementation level

The ProcessBase language consists of a core part and a system dependent part. The core part of the language must be provided by all implementations whereas the system dependent part may be implemented in different ways depending on the host system. The system dependent part includes persistence, IO, threads, strings, semaphores and mathematical functions. Libraries that implement these components on various platforms are supplied with the ProcessBase system, and are known as the Standard Libraries. They are described in a separate manual, the ProcessBase Standard Library Reference Manual.

Three sets of rules are used to define ProcessBase. The context free syntax of the languages is captured by using extended BNF. This context free syntax is then constrained by type rules into only allowing context sensitive constructions. The BNF and types are used in this manual. The meaning of every legal ProcessBase language construct is defined in terms of a set of code generation rules that describe the effect of the construct as a sequence of instructions to an abstract machine. The code generation rules are given in the ProcessBase Abstract Machine Manual.

As mentioned above, ProcessBase is the first in a family of languages. The reflective compiler is defined in terms of ProcessBase and implemented in it. Using that the hyper-code system will be added. The conceptual approach is that any language in the compliant architecture will be implemented by reflecting into ProcessBase itself. Thus a process modelling language or a language allowing polymorphic definition of code may be added as higher layers of the compliant architecture.

2 Context Free Syntax Specification

The formal definition of a programming language gives programmers a precise description from which to work as well as providing implementors with a reference model. There are two levels of definition, syntactic and semantic. This section deals with the formal syntactic rules used to define the context free syntax of the language. The context free syntactic rules are further qualified by a set of context sensitive type rules, given in Appendix II. Later, informal semantic descriptions of the syntactic categories will be given. The formal rules define the set of all syntactically legal ProcessBase programs, remembering that the meaning of any one of these programs is defined by the semantics.

To define the syntax of a language another notation, called a meta language, is required and in this case a variation of Backus-Naur form is used.

The syntax of ProcessBase is specified by a set of rules called *productions*. Each production specifies the manner in which a particular syntactic category (e.g. a clause) can be formed. Syntactic categories have names which are used in productions and are distinguished from names and reserved words in the language. The syntactic categories can be mixed in productions with terminal symbols which are actual symbols of the language itself. Thus, by following the productions until terminal symbols are reached, the set of legal programs can be derived.

The meta symbols, that is those symbols in the meta language used to describe the grammar of the language, include | which allows a choice in a production. The square brackets [and] are used in pairs to denote that an term is optional. When used with a *, a zero or many times repetition is indicated. The reader should not confuse the meta symbols |, *, [and] with the actual symbols and reserved words in ProcessBase. To help with this reserved words will appear in **bold** and symbols of ProcessBase will appear in 𝐨𝐮𝐭𝐥𝐢𝐧𝐞 𝐛𝐨𝐥𝐝. The names of the productions will appear in *italics*.

For example,

identifier ::= *letter* [*letter* | *digit* | _]*

indicates that an identifier can be formed as a letter, optionally followed by zero or many letters, digits or underbars.

The productions for ProcessBase are recursive, which means that there are an infinite number of legal ProcessBase programs. However, the syntax of ProcessBase can be described in about 45 productions.

The full context-free syntax of ProcessBase is given in Appendix I.

3 Types and Type Rules

The ProcessBase type system is based on the notion of types as a set structure imposed over the value space. Membership of the type sets is defined in terms of common attributes possessed by values, such as the operations defined over them. These sets or types partition the value space. The sets may be predefined, like **int**, or they may be formed by using one of the predefined type constructors, like **view**.

The constructors obey the *Principle of Data Type Completeness* [Str67, Mor79]. That is, where a type may be used in a constructor, any type is legal without exception. This has two benefits. Firstly, since all the rules are very general and without exceptions, a very rich type system may be described using a small number of defining rules. This reduces the complexity of the defining rules. The second benefit is that the type constructors are as powerful as is possible since there are no restrictions on their domain.

3.1 Universe of Discourse

The following base types are defined in ProcessBase:

1. The scalar data types are *int*, *real*, and *bool*.

2. Type *string* is the type of a character string; this type embraces the empty string and single characters.

3. Type *any* is an infinite union type; values of this type consist of a value of any type together with a representation of that type.

The following type constructors are defined in ProcessBase:

4. For any type T, *loc [T]* is the type of a location that contains a value of type T.

5. For any type t, $*t$ is the type of a vector with elements of type t.

6. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, *view* $[I_1: t_1,...,I_n: t_n]$ is the type of a view with fields I_i and corresponding types t_i, for i = 1..n and n ≥ 0.

7. For any types $t_1,...,t_n$ and t, *fun* $(t_1,...,t_n) \rightarrow t$ is the type of a procedure with parameter types t_i, for i = 1..n, where n ≥ 0, and result type t. The type of a result-less procedure is *fun* $(t_1,...,t_n)$.

8. For any types $t_1,...,t_n$ and t, *interrupt* $(t_1,...,t_n) \rightarrow t$ is the type of an interrupt with parameter types t_i, for i = 1..n, where n ≥ 0, and result type t. The type of a result-less interrupt is *interrupt* $(t_1,...,t_n)$.

9. For any types $t_1,...,t_n$ and t, *opcode* $[int,...,int](t_1,...,t_n) \rightarrow t$ (where the square brackets contain *m* occurrences of type *int*) is the type of an op-code with *m* op-code parameters and stack parameter types t_i, for i = 1..n, where m ≥ 0 and n ≥ 0, and result type t. The type of a result-less op-code is *opcode* $[int,...,int](t_1,...,t_n)$.

The world of data values is defined by the closure of rules 1 to 3 under the recursive application of rules 4 to 7.

In addition to the above, clauses which yield no value are of type **void**.

3.2 The Type Algebra

ProcessBase provides a simple type algebra that allows the succinct definition of types within programs. As well as the base types and constructors already introduced, types may be defined with the use of aliasing and recursive definitions.

3.2.1 Aliasing

Any legal type description may be aliased by an identifier to provide a shorthand for that type. For example

```
type ron is int
type man is view [age : int ; size : real]
```

After its introduction an alias may be used in place of the full type description.

3.2.2 Recursive Definitions

Further expressibility may be achieved in the type algebra by the introduction of recursive types. The reserved word **rec** introduced before a type alias allows instances of that alias to appear in the type definition. Mutually recursive types may also be defined by the grouping of aliases with ampersands. In this case, binding of identifiers within the mutual recursion group takes precedence over identifiers already in scope.

```
rec type intList is view [head : int; tail : realList]
        & realList is view [head: real; tail : intList]
```

3.3 Type Equivalence

Type equivalence in ProcessBase is based upon the meaning of types, and is independent of the way the type is expressed within the type algebra. Thus any aliases and recursion variables are fully factored out before equivalence is assessed. This style of type equivalence is normally referred to as structural equivalence.

The structural equivalence rules are as follows:

- Every base type is equivalent only to itself.
- For two constructed types to be equivalent, they must have the same constructor and be constructed over equivalent types.
- The bounds of a vector are not significant for type equivalence.
- For view constructors the labels are a significant part of the type, but their ordering is not.
- For procedure types, the parameter ordering is a significant part of the type, but parameter names are not.

ProcessBase has no subtyping or implicit coercion rules. Values may be substituted by assignment or parameter passing only when their types are known statically to be equivalent.

The types of all expressions in ProcessBase are inferred. There is no other type inference mechanism; in particular, the types of all procedure parameters and results must be explicitly stated by the programmer.

3.4 The Syntax of Types

The set of legal type strings in ProcessBase is expressed syntactically by the following production:

type	::=	**int** | **real** | **bool** | **string** | **any** | *identifier* | **loc** ⟦*type*⟧ | ❋*type* | **view** ⟦*labelled_type_list*⟧ | **fun** ⟪[*type_list*]⟫ [→ *type*]
type_list	::=	*type* [, *type*]*
labelled_type_list	::=	*identifier_list* : *type* [; *labelled_type_list*]

3.5 Typing Rules

The type rules of ProcessBase are used in conjunction with the context free syntax to determine the legal set of (type correct) programs. For this a method of specifying the type rules is required.

Before that, however, the concept of environments is introduced. Two kinds of environments are used, both of which are sets of bindings: one for value identifiers and one for type identifiers. π denotes the environments where value identifiers are bound to their types in the form of <x, T>, where x is an identifier and T is a type. τ stands for the environments in which type identifiers are bound to type expressions in the form <t, T>, where t is an identifier and T is a type. A_1::b::A_2 is used to represent a list A which contains a binding b. A++B is used to denote the concatenation of two lists of bindings A and B. Both π and τ are global environments and support block structure.

As introduced above bindings are represented as pairs and the notation < x, T > is used to denote a pair value consisting of x and T. (b_1, . . . , b_n) is a list containing bindings b_1 to b_n. The meta function *typeDecl* takes a list of bindings between type identifiers and type expressions and adds them to the environment τ. Similarly, the meta function *idDecl* takes a list of bindings between identifiers and types as its arguments and updates the environment π with the new bindings.

The typing rules use the structure of proof rules i.e.

$$\frac{A_1 \quad A_2 \quad \ldots \quad A_n}{B}$$

means that if A_i is true for i = 1, . . n then B is true. Each A_i and B may be of the form X ⊢ Y which, in the context of this document, is used to denote that Y is deducible from a collection of environments X. Thus, the type rule

$$\frac{\tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int}}{\tau, \pi \vdash e_1 + e_2 : \textbf{int}}$$

is read as "if expression e_1 is deduced to be of type **int** from environments τ and π and expression e_2 is deduced to be of type **int** from environments τ and π then the type of the expression $e_1 + e_2$ can be deduced to be of type **int** from environments τ and π ".

The typing rules make use of a set Type which is a set of strings defined from the production *type* in section 3.4. If a type S can be generated by this definition then S is a member of Type.

The type rules will be used throughout this manual, in conjunction with the context-free syntax rules, to describe the language. A complete set of type rules for ProcessBase is given in Appendix II.

3.6 First Class Citizenship

The application of the *Principle of Data Type Completeness* [Str67, Mor79] ensures that all data types may be used in any combination in the language. For example, a value of any data type may be a parameter to or returned from a procedure. In addition to this, there are a number of properties possessed by all values of all data types that constitute their civil rights in the language and define first class citizenship. All values of data types in ProcessBase have first class citizenship.

The additional civil rights that define first class citizenship are:

- the right to be declared,
- the right to be assigned,
- the right to have equality defined over them, and,
- the right to persist.

4 Literals

Literals are the basic building blocks of ProcessBase programs that allow values to be introduced. A literal is defined by:

literal ::= *int_literal* | *real_literal* | *bool_literal* | *string_literal* | *view_literal* | *proc_literal*

4.1 Integer Literals

These are of type **int** and are defined by:

int_literal ::= [*add_op*] *digit* [*digit*]*
add_op ::= + | -

$$\frac{n \in \text{Integer}}{n : \textbf{int}}$$ [intLiteral]

An integer literal is one or more digits optionally preceded by a sign. For example,

1	0	1256	-8797

4.2 Real Literals

These are of type **real** and are defined by

real_literal ::= *int_literal*.[*digit*]*[e *int_literal*]

$$\frac{r \in \text{Real}}{r : \textbf{real}}$$ [realLiteral]

Thus, there are a number of ways of writing a real literal. For example,

1.2	3.1e2	5.e5
1.	3.4e-2	3.4e+4

3.1e-2 means 3.1 times 10 to the power -2 (i.e. 0.031)

4.3 Boolean Literals

There are two literals of type **bool**: **true** and **false**. They are defined by

bool_literal ::= **true** | **false**

$$\frac{b \in \text{Boolean}}{b : \textbf{bool}}$$ [boolLiteral]

4.4 String Literals

A string literal is a sequence of characters in the character set (ASCII) enclosed by double quotes. The syntax is

string_literal ::= "[*char*]*"

$$\frac{s \in \text{ASCII character set}}{s : \textbf{string}} \quad \text{[strLiteral]}$$

The empty string is denoted by "". Examples of other string literals are:

"This is a string literal", and, "I am a string"

The programmer may wish to have a double quote itself inside a string literal. This requires using a single quote as an escape character and so if a single or double quote is required inside a string literal it must be preceded by a single quote. For example,

"a"""" has the value a", and, "a"" has the value a'.

There are a number of other special characters that may be used inside string literals. They are:

'b	backspace	ASCII code 8
't	horizontal tab	ASCII code 9
'n	newline	ASCII code 10
'p	newpage	ASCII code 12
'o	carriage return	ASCII code 13

4.5 View Literals

There is one literal for each view constructor type. It is used to ground recursion in view types.

view_literal ::= **nil** (*type*)

$$\frac{}{\tau_1 ::< t, \textbf{ view } [l_1 : T_1; \ldots; l_n : T_n] >:: \tau_2, \pi \ \mathsf{h}\ \textbf{nil}\ (t) : \textbf{view } [l_1 : T_1; \ldots; l_n : T_n]} \quad \text{[viewLiteral]}$$

4.6 Procedure Literals

A procedure is introduced into a program by its literal value. It is defined by:

proc_literal ::= **fun** ([*labelled_type_list*]) [→ *type*]; *clause*
labelled_type_list ::= *identifier_list* : *type* [; *labelled_type_list*]

$$\frac{\tau, \pi_1 ::< x_1, T_1 >:: \pi_2 :: \ldots ::< x_n, T_n >:: \pi_{n+1} \ \mathsf{h}\ e : S}{\tau, \pi \ \mathsf{h}\ \textbf{fun}(x_1 : T_1, \ldots, x_n : T_n) \rightarrow S ; e : \textbf{fun}(T_1, \ldots, T_n) \rightarrow S} \quad \text{[procLiteral]}$$

For example,

```
fun (n : int) → int ; n
```

is a procedure literal.

5 Expressions and Operators

5.1 Evaluation Order

The order of execution of a ProcessBase program is strictly from left to right and top to bottom except where the flow of control is altered by one of the language clauses. This rule becomes important in understanding side-effects in the store. Parentheses in expressions can be used to override the precedence of operators.

5.2 Parentheses

In the syntactic description there are two productions:

clause ::= ... | *E*
E ::= ... | ((*clause*))

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash (\ e\ ) : T} \qquad \text{[brackets]}$$

These rules allow expressions in ProcessBase to be written within parentheses. The effect of this is to alter the order of evaluation so that the expressions in parentheses are evaluated first. For example:

3 * (2 - 3)

evaluates to -3 and not 3.

5.3 Boolean Expressions

Values of type **bool** in ProcessBase can have the value true or false. There are only two boolean literals, **true** and **false**, and three operators. There is one boolean unary operator, ~, and two boolean binary operators, **and** and **or**. They are defined by the truth table below:

a	b	~a	a **or** b	a **and** b
true	**false**	**false**	**true**	**false**
false	**true**	**true**	**true**	**false**
true	**true**	**false**	**true**	**true**
false	**false**	**true**	**false**	**false**

The syntax rules for boolean expressions are:

E ::= ~ *E* | *E* **or** *E* | *E* **and** *E*

$$\frac{\tau, \pi \vdash e : \textbf{bool}}{\tau, \pi \vdash \sim e : \textbf{bool}} \qquad \text{[negation]}$$

$$\frac{\tau, \pi \vdash e_1 : \textbf{bool} \quad \tau, \pi \vdash e_2 : \textbf{bool}}{\tau, \pi \vdash e_1 \ \textbf{and}\ e_2 : \textbf{bool}} \qquad \text{[and]}$$

$$\frac{\tau, \pi \vdash e_1 : \textbf{bool} \quad \tau, \pi \vdash e_2 : \textbf{bool}}{\tau, \pi \vdash e_1 \ \textbf{or}\ e_2 : \textbf{bool}} \qquad \text{[or]}$$

The precedence of the operators is important and is defined in descending order as:

~
and
or

Thus,

~a **or** b **and** c

is equivalent to

(~a) **or** (b **and** c)

The evaluation of a boolean expression in ProcessBase is non-strict. That is, in the left to right evaluation of the expression, no more computation is performed on the expression than is necessary. For example,

true or *expression*

gives the value **true** without evaluating *expression* and

false and *expression*

gives the value **false** without evaluating *expression*.

5.4 Comparison Operators

Expressions of type **bool** can also be formed by some other binary operators. For example, a = b is either **true** or **false** and is therefore boolean. These operators are called the comparison operators and are:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
~=	not equal to

The syntactic rules for the comparison operators are:

E ::= *E rel_op E*
rel_op ::= *eq_op* | *co_op*
eq_op ::= = | ~=
co_op ::= < | <= | > | >=

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 = e_2 : \textbf{bool}} \text{ [equality]}$$

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 \sim= e_2 : \textbf{bool}} \text{ [nonEq]}$$

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 < e_2 : \textbf{bool}}$$ [less]

where T ∈ { **int**, **real**, **string** }

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 <= e_2 : \textbf{bool}}$$ [lessEq]

where T ∈ { **int**, **real**, **string** }

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 > e_2 : \textbf{bool}}$$ [greater]

where T ∈ { **int**, **real**, **string** }

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 >= e_2 : \textbf{bool}}$$ [greaterEq]

where T ∈ { **int**, **real**, **string** }

Note that the operators <, <=, > and >= are defined on integers, reals and strings whereas = and ~= are defined on all ProcessBase data types. The interpretation of these operations is given with each data type as it is introduced.

Equality for types other than base types is defined as identity.

5.5 Arithmetic Expressions

Arithmetic may be performed on data values of type **int** and **real**. The syntax of arithmetic expressions is:

E	::=	*add_op E* |*E add_op E* | *E mult_op E*
add_op	::=	+ | -
mult_op	::=	*int_mult_op* | *real_mult_op* | ...
int_mult_op	::=	* | **div** | **rem**
real_mult_op	::=	* | /

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash +e : T}$$ [plus]

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash -e : T}$$ [minus]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 + e_2 : T}$$ [add]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 - e_2 : T}$$ [subtract]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 * e_2 : T}$$ [times]

$$\frac{\tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int}}{\tau, \pi \vdash e_1 \ \texttt{div} \ e_2 : \textbf{int}} \quad \text{[intDiv]}$$

$$\frac{\tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int}}{\tau, \pi \vdash e_1 \ \texttt{rem} \ e_2 : \textbf{int}} \quad \text{[intRem]}$$

$$\frac{\tau, \pi \vdash e_1 : \textbf{real} \quad \tau, \pi \vdash e_2 : \textbf{real}}{\tau, \pi \vdash e_1 \ / \ e_2 : \textbf{real}} \quad \text{[realDiv]}$$

The operators mean:

+	addition
-	subtraction
*	multiplication
/	real division
div	integer division throwing away the remainder
rem	remainder after integer division

In both **div** and **rem** the result is negative only if exactly one of the operands is negative.

Some examples of arithmetic expressions are

a + b	3 + 2	1.2 + 0.5	-2.1 + a / 2.0

The language deliberately does not provide automatic coercion between integers and reals, but conversion procedures are defined in the ProcessBase Standard Library «Morrison, 1999 #2234».

5.6 Arithmetic Precedence Rules

The order of evaluation of an expression in ProcessBase is from left to right and based on the precedence table:

*	/	**div**	**rem**
+	-		

That is, the operations *, /, **div**, **rem** are always evaluated before + and -. However, if the operators are of the same precedence then the expression is evaluated left to right. For example,

6 **div** 4 **rem** 2 gives the value 1

Brackets may be used to override the precedence of the operator or to clarify an expression. For example,

3 * (2 - 1) yields 3 not 5

5.7 String Expressions

The string operator, ++, concatenates two operand strings to form a new string. For example,

"abc" ++ "def"

results in the string

"abcdef"

The syntax rule is:

E::= *E* [++ *E*]*

$$\frac{\tau, \pi \vdash e_1 : \textbf{string} \quad \tau, \pi \vdash e_2 : \textbf{string}}{\tau, \pi \vdash e_1 \ ++ \ e_2 : \textbf{string}} \qquad \text{[concat]}$$

A new string may be formed by selecting a substring of an existing string. For example, if s is the string "abcdef" then s (3 | 2) is the string "cd". That is, a new string is formed by selecting 2 characters from s starting at character 3. The syntax rule is:

E ::= *E* (*clause* | *clause*)

$$\frac{\tau, \pi \vdash e : \textbf{string} \quad \tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int}}{\tau, \pi \vdash e \ (e_1 \mid e_2) : \textbf{string}} \qquad \text{[substr]}$$

For the purposes of substring selection the first character in a string is numbered 1. The selection values are the start position and the length respectively.

To compare two strings, the characters are compared in pairs, one from each string, from left to right. Two strings are considered equal only if they have the same characters in the same order and are of the same length, otherwise they are not equal.

The characters in a string are ordered according to the ASCII character code. Thus,

"a" < "z"

is true.

The *null* string is less than any other string. Thus the less-than relation can be resolved by taking the characters pair by pair in the two strings until one is found to be less than the other. When the strings are not of equal length then they are compared as above and then the shorter one is considered to be less that the longer. Thus,

"abc" < "abcd"

The other relations can be defined by using = and <.

5.8 Precedence Table

The full precedence table for operators in ProcessBase is:

/	*	**div**	**rem**
+	-	++	
~			
=	~=	<	<=	>	>=
and			
or			

6 Locations

Values may be stored in locations and subsequently retrieved. The constructor **loc** creates a location and initialises the value in it. The operator ' (dereference) retrieves the value from the location. Since locations are values in ProcessBase they may also be stored in locations. The syntactic rules are:

E::= **loc** (*clause*) | '*clause*

$$\frac{\tau,\ \pi\ \vdash e : T}{\tau,\ \pi\ \vdash \mathbf{loc}(\ e\ ) : \mathbf{loc}[\ T\ ]} \quad \text{[locValue]}$$

$$\frac{\tau,\ \pi\ \vdash e : \mathbf{loc}[\ T\ ]}{\tau,\ \pi\ \vdash\ 'e : T} \quad \text{[locDeref]}$$

For example, if *a* is of type **loc** [**int**] with the value **loc** (3) then *'a* has the value 3.

6.1 Equality and Equivalence

Two locations are equal if they have the same identity, that is, the same location. Two locations are type equivalent if they have equivalent content types. Notice therefore that

loc (3) = **loc** (3)

will yield the result false.

7 Declarations

7.1 Identifiers

In ProcessBase, an identifier may be bound to a data value, a procedure parameter, a view field, or a type. An identifier may be formed according to the syntactic rule

identifier ::= *letter* [*id_follow*]
id_follow ::= *letter* [*id_follow*] | *digit* [*id_follow*] | _ [*id_follow*]

That is, an identifier consists of a letter followed by any number of underscores, letters or digits. The following are legal ProcessBase identifiers:

x1 ronsValue look_for_Record1 Ron

Note that case is significant in identifiers.

The use of an identifier is governed by the syntactic rule

E ::= *identifier*

The type rule states that the type of an identifier can be deduced from the value environment π.

$$\frac{}{\tau,\ \pi_1 :: < x,\ T > :: \pi_2\ \vdash x : T} \quad \text{[id]}$$

7.2 Declaration of Value Identifiers

Before an identifier can be used in ProcessBase, it must be declared. The action of declaring a data value associates an identifier with a typed value.

When introducing an identifier, the programmer must indicate the identifier and its value. Identifiers are declared using the following syntax:

value_decl ::= **let** *identifier* ← *clause*

$$\frac{\tau,\ \pi\ \vdash e : T}{\tau,\ \pi\ \vdash \mathbf{let}\ x\ \leftarrow\ e : \mathbf{void}\quad \text{idDecl}(\ (\ <\ x,\ T\ >\ )\ )} \quad \text{[valueDecl]}$$

An identifier is declared by

let identifier ← *clause*

For example,

```
let a ← 1
```

introduces an integer identifier with value 1. Notice that the compiler deduces the type.

Identifiers can also be declared for locations, for example,

```
let discrim ← loc (b * b - 4.0 * a * c)
```

introduces a real number location with the calculated value. The value in the location may be updated by assignment.

7.3 Declaration of Types

Type names may be declared by the user in ProcessBase. The name is used to represent a set of values drawn from the value space and may be used wherever a type identifier is legal. The syntax of type declarations is:

type_decl ::= **type** *type_init* | **rec type** *type_init* [& *type_init*]*
type_init ::= *identifier* **is** *type*

$$\frac{\tau \vdash T \in \text{Type}}{\tau \vdash \textbf{type t is T} : \textbf{void} \quad \text{typeDecl}(\ (\ <\ t,\ T\ >\ )\ )} \qquad \text{[typeDecl]}$$

Thus,

```
type al is bool
```

is a type declaration aliasing the identifier *al* with the boolean type. They are the same type and may be used interchangeably.

7.4 Sequences

A sequence is composed of any combination, in any order, of declarations and clauses. The type of the sequence is the type of the last clause in the sequence. Where the sequence ends with a declaration, which by definition is of type *void*, the sequence is of type *void*. If there is more than one clause in a sequence then all but the last must be of type *void*.

sequence ::= *declaration* [; *sequence*] | *clause* [; *sequence*]

$$\frac{\tau,\ \pi \vdash A_1 : \textbf{void} \quad \text{Decl}_1 \quad \tau + +\Omega_1,\ \pi + +\Psi_1 \vdash A_2 : T}{\tau,\ \pi \vdash A_1\ ;\ A_2 : T} \qquad \text{[seq]}$$

where Decl_1 stands for typeDecl(Ω_1) and idDecl(Ψ_1), and A_1 and A_2 stand for any constructs in the language

7.5 Brackets

Brackets are used to make a sequence of clauses and declarations into a single clause. There are two forms, which are:

```
begin
        sequence
end
```

and

{*sequence*}

$$\frac{\tau,\ \pi\ \mathsf{h}\ s : T}{\tau,\ \pi\ \mathsf{h}\ \mathbf{begin}\ \ s\ \ \mathbf{end} : T} \quad \text{[beginEnd]}$$

$$\frac{\tau,\ \pi\ \mathsf{h}\ s : T}{\tau,\ \pi\ \mathsf{h}\ \{\ \ s\ \ \} : T} \quad \text{[\{\}]}$$

{ and } allow a sequence to be written clearly on one line as a clause. For example,

```
let i ← loc (2)
for j ← 1 to 5 do {i := 'i * 'i ; writeInt ('i)}
```

However, if the sequence is longer than one line, the first alternative gives greater clarity. Non-void sequences are sometimes called block expressions.

7.6 Scope Rules

The scope of an identifier is limited to the rest of the sequence following the declaration. This means that the scope of an identifier starts immediately after the declaration and continues up to the next unmatched } or **end**. If the same identifier is declared in an inner sequence, then while the inner name is in scope the outer one is not.

7.7 Recursive Value Declarations

It is sometimes necessary to define values recursively. For example, the following defines a recursive version of the factorial procedure:

```
rec let factorial ←   fun (n : int) → int
                      if n = 0 then 1 else n * factorial (n - 1)
```

The effect of the recursive declaration is to allow the identifier to enter scope immediately. That is, before the declaration clause and not immediately after it, as is the case with non-recursive declarations. Thus, the identifier *factorial* used in the procedure is the same as, and refers to, the one being defined.

Where there is more than one identifier being declared, all the identifiers come into scope at the same time. That is, all the names are declared first and then are available for the initialising clauses.

The initialising clauses for recursive declarations are restricted to literal values.

The full syntax of value declarations is:

value_decl	::=	**let** *value_init* |
		rec let *rec_value_init* [& *rec_value_init*]*
value_init	::=	*identifier* ← *clause*
rec_value_init	::=	*identifier* ← *literal*

$$\frac{\tau,\ \pi'\ \mathsf{h}\ e_1 : T_1 \qquad \tau,\ \pi'\ \mathsf{h}\ e_2 : T_2}{\tau,\ \pi\ \mathsf{h}\ \mathbf{rec\ let}\ x_1\ \leftarrow\ e_1\ \&\ x_2\ \leftarrow\ e_2 : \mathbf{void} \quad \text{Decl}} \quad \text{[recValDecl]}$$

where π' stands for $\pi_1{::}< x_1, T_1 >{::}\pi_2{::}< x_2, T_2 >{::}\pi_3$ and Decl stands for idDecl(($< x_1, T_1 >$, $< x_2, T_2 >$))

7.8 Recursive Type Declarations

The full syntax of type declarations is:

type_decl ::= **type** *type_init* | **rec type** *type_init* [& *type_init*]*
type_init ::= *identifier* **is** *type*

$$\frac{\tau \vdash T_1 \in \text{Type} \quad \tau \vdash T_2 \in \text{Type}}{\tau' \vdash \textbf{rec type } t_1 \textbf{ is } T_1 \ \& \ t_2 \textbf{ is } T_2 : \textbf{void} \quad \text{typeDecl}(\ (< t_1, T_1 >, < t_2, T_2 >)\ )}$$

where τ' stands for $\tau_1 :: < t_1, T_1 > :: \tau_2 :: < t_2, T_2 > :: \tau_3$ [recTypeDecl]

For example, the following recursive type definition:

```
rec type intList is view [head : int; tail : realList]
        & realList is view [head: real; tail : intList]
```

defines types for lists whose head elements are either integer or real and whose tail elements are the other type of list.

8 Clauses

Expressions are clauses which allow the operators in the language to be used to produce data values. There are other kinds of clauses in ProcessBase which allow the data values to be manipulated and which provide control over the flow of the program.

8.1 Assignment Clause

The assignment clause has the following syntax:

clause ::= *name* :≔ *clause*

$$\frac{\tau,\ \pi\ \mathsf{h}\ e_2 : T \qquad \tau,\ \pi\ \mathsf{h}\ e_1 : \textbf{loc}[\ T\ ]}{\tau,\ \pi\ \mathsf{h}\ e_1\ :=\ e_2 : \textbf{void}} \quad \text{[assign]}$$

For example,

```
let discriminant ← loc (0.0)
discriminant := b * b - 4.0 * a * c
```

gives *discriminant* the value of the expression on the right. The clause alters the value in the location denoted by the identifier.

The semantics of assignment is defined in terms of equality. The clause,

a := b

where *a* and *b* are both identifiers, implies that after execution *'a = b* will be true. Thus, as will be seen later, assignment for scalar types means value assignment and for constructed types it means pointer assignment.

8.2 if Clause

There are two forms of the **if** clause defined by:

if *clause* **do** *clause* |
if *clause* **then** *clause* **else** *clause*

$$\frac{\tau,\ \pi\ \mathsf{h}\ e : \textbf{bool} \qquad \tau,\ \pi\ \mathsf{h}\ e_1 : \textbf{void}}{\tau,\ \pi\ \mathsf{h}\ \texttt{if}\ e\ \textbf{do}\ e_1 : \textbf{void}} \quad \text{[ifDo]}$$

$$\frac{\tau,\ \pi\ \mathsf{h}\ e : \textbf{bool} \qquad \tau,\ \pi\ \mathsf{h}\ e_1 : T \qquad \tau,\ \pi\ \mathsf{h}\ e_2 : T}{\tau,\ \pi\ \mathsf{h}\ \texttt{if}\ e\ \textbf{then}\ e_1\ \texttt{else}\ e_2 : T} \quad \text{[ifThen]}$$

In the single armed version, if the condition after the **if** is true, then the clause after the **do** is executed. For example, in the clause

```
if 'a < b do a := 3
```

the value 3 will be assigned to *a,* if the value in *a* is smaller than *b* before the **if** clause is executed.

The second version allows a choice between two actions to be made. If the first clause is **true**, then the second clause is executed, otherwise the third clause is executed. Notice that the second and third clauses are of the same type and the result is of that type. The following contains two examples of **if** clauses:

```
if 'x = 0 then y := 1 else x := 'y - 1
let temp ← if a < b then 1 else 5
```

8.3 while ... do Clause

The **while** clause allows loops to be constructed with the test at the start of the loop. The syntax is:

while *clause* **do** *clause*

$$\frac{\tau, \pi \vdash e : \textbf{bool} \quad \tau, \pi \vdash e_1 : \textbf{void}}{\tau, \pi \vdash \textbf{while } e \textbf{ do } e_1 : \textbf{void}} \qquad \text{[while]}$$

The loop is executed until the boolean clause is **false** and is used to perform a loop zero or many times.

An example of the **while** ... **do** clause is

```
let factorial ← loc (1) ; let i ← loc (0)
while 'i < 8 do
begin
        writeString ("Factorial "); writeInt ('i); writeString (" is ")
        writeInt ('factorial); writeString ("'n")
        i := 'i + 1 ; factorial := 'factorial * 'i
end
```

8.4 for Clause

The **for** clause is included in the language as syntactic sugar where there is a fixed number of iterations defined at the initialisation of the loop. It is defined by:

for *identifier* ← *clause* **to** *clause* [**by** *clause*] **do** *clause*

$$\frac{\tau, \pi_1 ::< \text{i}, \textbf{int} >:: \pi_2 \vdash e : \textbf{void} \quad \tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int} \quad \tau, \pi \vdash e_3 : \textbf{int}}{\tau, \pi \vdash \textbf{for i} \leftarrow e_1 \textbf{ to } e_2 \textbf{ by } e_3 \textbf{ do } e : \textbf{void}} \qquad \text{[for]}$$

in which the clauses are: the initial value, the limit, the increment and the clause to be repeated, respectively. The first three are of type **int** and are calculated only once at the start. The **by** clause may be omitted where the increment is 1. The identifier, known as the control constant, is in scope within the void clause, taking on the range of values successively defined by initial value, increment and limit. That is, the control constant is considered to be declared at the start of the repetition clause. The repetition clause is executed as many times as necessary to complete the loop and each time it is, the control constant is initialised to a new value, starting with the initial loop value, changing by the increment until the limit is reached. An example of a **for** clause is:

```
let factorial ← loc (1) ; let n ← 8
for i ← 1 to n do factorial := 'factorial * i
```

With a positive increment, the **for** loop terminates when the control constant is initialised to a value greater than the limit. With a negative increment, the **for** loop terminates when the control constant is initialised to a value less than the limit.

9 Procedures

9.1 Procedure Calls

Procedures in ProcessBase constitute abstractions over expressions, if they return a value, and over clauses of type **void** if they do not. In accordance with the *Principle of Correspondence* [Str67], any method of introducing a name in a declaration has an equivalent form as a parameter.

Thus, in declarations of data values, giving a name an initial value is equivalent to assigning the actual parameter value to the formal parameter. Since this is the only type of declaration for data values in the language, it is also the only parameter passing mode and is commonly known as *call by value*.

Like declarations, the formal parameters representing data values must have a name and a type. A procedure which returns a value must also specify its return type. The scope of the formal parameters is from their declaration to the end of the procedure clause.

The integer identity procedure, called *int_id*, may be declared by:

```
let int_id ← fun (n : int) → int; n
```

The syntax of a procedure call is:

expression ⟪[*clause_list*]⟫

$$\frac{\forall i \in \{1...n\}\ (\tau, \pi \vdash e_i : T_i) \quad \tau, \pi \vdash e : \mathbf{fun}(T_1, \ldots, T_n) \rightarrow S}{\tau, \pi \vdash e(e_1, \ldots, e_n) : S} \qquad \text{[procApp]}$$

There must be a one-to-one correspondence between the actual and formal parameters and their types. Thus, to call the integer identity procedure given above, the following could be used,

```
int_id (42)
```

which will evaluate to the integer 42.

The type of *int_id* is written **fun** (**int**) → **int**.

9.2 Recursive Declarations

Recursive and mutually recursive declarations of procedures are allowed in ProcessBase. For example,

```
rec let tak ← fun (x, y, z : int) → int
                  if x <= y then z else tak (   tak (x - 1, y, z),
                                                tak (y - 1, z, x),
                                                tak (z - 1, x, y) )
```

declares the recursive Takeuchi procedure. Mutually recursive procedures may also be defined. For example,

```
rec let expression ← fun () ; while have ("or") do exp1 ()
& exp1 ← fun () ; while have ("and") do exp2 ()
& exp2 ← fun ()
            if symb = "identifier" then next_symbol ()
            else {mustbe ("(") ; expression () ; mustbe (")")}
```

declares three mutually recursive procedures.

9.3 Equality and Equivalence

Two procedures are equal in ProcessBase if and only if their values are derived from the same evaluation of the same procedure expression. For the cognoscenti, this means that they have the same closure.

In common with all aggregate values in ProcessBase, equality means identity.

Two procedure types are structurally equivalent if they have the same parameter types in one-one correspondence and the same result type.

10 Aggregates

ProcessBase allows the programmer to group together data values into larger aggregate values which may then be treated as single values. There are two such value types in ProcessBase: vectors and views. If the constituent values are of the same type, a vector may be used and a view otherwise. Vectors and views have the same civil rights as any other data value in ProcessBase.

All aggregate data values in ProcessBase have pointer semantics. That is, when an aggregate data value is created, a pointer to the aggregate that make up the value is also created. The value is always referred to by the pointer which may be passed around by assignment and tested for equality.

10.1 Vectors

10.1.1 Creation of Vectors

A vector provides a method of grouping together values of the same type. Since ProcessBase does not allow uninitialised locations, all the initial values of the elements must be specified. The syntax is:

vector_constructor ::= **vector** *vector_element_init*
vector_element_init ::= *clause* **to** *clause* **using** *clause* |
@clause **of** ⟦*clause* [, *clause*]*⟧

$$\frac{\tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int} \quad \tau, \pi \vdash e : \textbf{fun}(\ \textbf{int}\ ) \rightarrow T}{\tau, \pi \vdash \textbf{vector}\ e_1\ \textbf{to}\ e_2\ \textbf{using}\ e : *T} \quad \text{[vecValue1]}$$

$$\frac{\tau, \pi \vdash e : \textbf{int} \quad \forall i \in \{1...n\}\ (\tau, \pi \vdash e_i : T)}{\tau, \pi \vdash \textbf{vector}\ @\ e\ \textbf{of}\ [\ e_1, \ldots, e_n\ ] : *T} \quad \text{[vecValue2]}$$

For example,

```
let abc ← vector @1 of [ 1, 2, 3, 4 ]
```

declares *abc* to be a vector of integers, whose type is written as ***int**, with lower bound 1 and elements initialised to 1, 2, 3 and 4.

Multi-dimensional vectors, which are not necessarily rectangular, can also be created. For example,

```
let Pascal = vector @1 of [
                    vector @1 of [ 1 ],
                    vector @1 of [ 1, 1 ],
                    vector @1 of [ 1, 2, 1 ],
                    vector @1 of [ 1, 3, 3, 1 ],
                    vector @1 of [ 1, 4, 6, 4, 1 ],
                    vector @1 of [ 1, 5, 10, 10, 5, 1 ] ]
```

Pascal is of type ****int**.

A second form of vector initialisation is provided to allow the elements of a vector to be initialised by a function over the index. For example,

```
let squares ← fun (n : int) → int; n * n
let squares_vector ← vector 1 to 10 using squares
```

In the initialisation, the procedure *squares* is called for every index of the vector in order from the lower to upper bound. The corresponding element is initialised to the result of its own index being passed to the procedure. In the above case, the vector *squares_vector* has elements 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100.

The initialising procedure must be of type

fun (**int**) → t

and the resulting vector is of type **t*.

10.1.2 upb and lwb

It is often necessary to interrogate a vector to find its bounds. The operations *upb* and *lwb* are provided in ProcessBase for this purpose:

$$\frac{\tau, \pi \vdash e : *T}{\tau, \pi \vdash \texttt{lwb(}\ e\ \texttt{)} : \textbf{int}} \quad \text{[lwb]}$$

$$\frac{\tau, \pi \vdash e : *T}{\tau, \pi \vdash \texttt{upb(}\ e\ \texttt{)} : \textbf{int}} \quad \text{[upb]}$$

10.1.3 Indexing

To obtain the elements of a vector, indexing is used. For vectors, the index is always an integer value. The syntax is:

E ::= *E* ⟪*clause_list*⟫

$$\frac{\tau, \pi \vdash e : *\ldots *T \quad \tau, \pi \vdash e_1 : \textbf{int} \ \ldots \ \tau, \pi \vdash e_n : \textbf{int}}{\tau, \pi \vdash \texttt{e(}\ e_1\texttt{, . . . ,}\ e_n\ \texttt{)} : T} \quad \text{[vecProj]}$$

For example,

a (3 + 4)

selects the element of the vector *a* which is associated with the index value 7. Multi-dimensional vectors may be indexed by using commas to separate the indices.

10.1.4 Equality and Equivalence

Two vectors are equal if they have the same identity, that is, the same pointer. Two vectors are type equivalent if they have equivalent element types. Notice that the bounds are not part of the type.

10.2 Views

10.2.1 Creation of View

Values of different types can be grouped together into a view. The fields of a view have identifiers that are unique within that view. The views are sets of labelled cross products from the value space. Views are created using a type identifier. The syntax of view types is:

type ::= **view** ⟦*labelled_type_list*⟧
labelled_type_list ::= *identifier_list* ⁝ *type* [⨾ *labelled_type_list*]

For example, a view type may be declared as follows:

```
type person is view [name : string ; age, height : int]
```

This declares a view type, *person*, with three fields of type **string**, **int** and **int**, with labels: *name*, *age* and *height* respectively.

A view may be created by the following syntax:

E ::= **view** (⦇*value_init_list*⦈)

value_init_list ::= *value_init* [, *value_init*]

value_init ::= *identifier* ← *clause*

$$\frac{\forall i \in \{ 1...n \} (\tau, \pi \mathrel{h} e_i : T_i)}{\tau, \pi \mathrel{h} \textbf{view}(l_1 \leftarrow e_1, \ldots, l_n \leftarrow e_n) : T}$$

where T stands for **view**[$l_1 : T_1, \ldots, l_n : T_n$] [viewValue]

For example,

```
let ron ← view (name ← "Ronald Morrison", age ← 42, height ← 175)
```

creates a view with field labels *name, age* and *height* and with field values “Ronald Morrison”, 42 and 175 respectively.

10.2.2 Indexing

To obtain a field of a view, the field identifier is used as an index. The syntax is

E ::= *clause*.*identifier*

$$\frac{\tau, \pi \mathrel{h} e : \textbf{view}[\ldots; l : T; \ldots]}{\tau, \pi \mathrel{h} e.l : T} \qquad \text{[viewDeref]}$$

For example, if *ron* is declared as above, then,

ron.age

yields 42. For the indexing operation to be legal, the view must contain a field with that identifier.

Field identifiers, when used as indices, are only in scope after the dot following a view expression. Thus these identifiers need only be unique within each view type.

10.2.3 Equality and Equivalence

Two views are equal if they have the same identity (pointer).

The type of a view is the set of the field identifier-type pairs. Thus the view *ron* has type:

> **view** [name : **string** ; age : **int** ; height : **int**]

Two views have equivalent types when the types have the same set of identifier-type pairs for the fields. Note that the order of the fields is unimportant.

11 Type any

Type **any** is the type of the union of all values in ProcessBase. Values must be explicitly injected into and projected from type **any**. Both of these operations are performed dynamically and, in particular, the projection from **any** to another type involves a dynamic type check. We have argued elsewhere [ABC+83] that such a type check is required to support the binding of independently prepared programs and data in a type secure persistent object store.

11.1 Injection into Type any

Values may be injected into type **any** by the following syntax:

any ⟪*clause*⟫

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash \textbf{any}(\ e\ ) : \textbf{any}} \qquad \text{[anyInj]}$$

For example,

```
let int_any ← any (-42)
```

which declares *int_any* to be the integer value -42 injected into type **any**.

Values of type **any** may be passed as parameters. For example, the following is an identity procedure for type **any**.

```
let id_any ← fun (x : any) → any ; x
```

Thus polymorphic procedures may be written by using type **any** and injecting the parameters into **any** before the call and projecting the results after the call.

11.2 Projection from Type any

Values may be projected from type **any** by use of the **project** clause.

project *clause* **as** *identifier* **onto** *project_list* **default** : *clause*
project_list ::= *type_id* : *clause* ; [*project_list*]

$$\frac{\tau, \pi \vdash e : \textbf{any} \quad T_1, \ldots, T_n \in \text{Type} \quad \forall i \in \{1..n\}\ (\tau, \pi' \vdash e_i : T)}{\tau, \pi \vdash \textbf{project}\ e\ \textbf{as}\ x\ \textbf{onto}\ \text{project\_list}\ ;\ \textbf{default}\ e_{n+1} : T}$$

where π' stands for π_1::< x, T_i >::π_2 and project_list stands for $T_1 : e_1 ; \ldots ; T_n : e_n$

[anyProj]

The projected value is bound to the identifier following the **as**. The scope of the identifier is the clauses on the right hand side of the colons. This mechanism prevents side effects on the projected value inside the evaluation of the right hand side clauses and allows for static type checking therein. For projection, the type is compared to each of the types on the left hand side of the colons. The first match causes the corresponding clause on the right hand side to be executed. Within the clause, the identifier has the type of the projected value. After execution of the **project** clause, control passes to the clause following the **project** clause.

An example of projection is:

```
let get_type ← fun (x : any) → string
                    project x as X onto
                         int    :    "type is integer"
                         real   :    "type is a real"
                    default     :    "type is neither integer nor real"
```

11.3 Equality and Equivalence

Two values of type **any** are equal if and only if they can be projected onto equivalent types and the projected values are equal.

All values of type **any** are type equivalent.

12 Exceptions, Interrupts and Down-calls

12.1 Exceptions

When an error occurs during execution of a ProcessBase program, an exception is raised and control is passed to an exception handler. The system supplies a default exception handler; additional application-specific handlers may be defined as required. Exceptions may be raised either automatically, due to an error arising during the execution of a built-in ProcessBase operation (a standard exception), or explicitly due to an application-specific error condition. The action of the default exception handler is to abort the thread in which it occurs, and to print the name and description of the exception to the standard output if supported on the current platform.

The table below shows the standard exceptions that may be raised:

name	description	circumstances
"arithmetical"	description of the attempted operation and parameters	if an arithmetical error occurs
"dereference"	description of the attempted operation and parameters	if an attempt is made to dereference a null view
"string"	description of the attempted operation and parameters	if an error occurs during a string operation
"vector"	description of the attempted operation and parameters	if an error occurs during a vector operation

Exceptions are represented by instances of the view type *Exception*, defined in the Exception Standard Library:

```
type Exception is view [name, description : string]
```

The default exception handling behaviour may be over-ridden by specifying an explicit exception handler for a clause. The reserved words **handle exception** introduce a local name for the exception instance, which is in scope for the exception handler clause. The syntax is:

clause ::= **handle exception** *identifier* **using** *clause* **in** *clause*

$$\frac{\tau, \pi_1 ::< x, \text{Exception} >:: \pi_2 \;\; h\, e_1 : T \quad \tau, \pi \; h\, e_2 : T}{\tau, \pi \; h\, \textbf{handle exception}\; x\; \textbf{using}\; e_1\; \textbf{in}\; e_2 : T} \qquad \text{[handleException]}$$

An exception may also be raised explicitly using a **raise** clause:

clause ::= **raise** *clause*

$$\frac{\tau, \pi \; h\, e : \text{Exception}}{\tau, \pi \; h\, \textbf{raise}\; e : \textbf{void}} \qquad \text{[raise]}$$

The example below shows both exception handling and raising:

```
handle exception e using
        if e.name = "arithmetical" then
                writeString ("arithmetical error: " ++  e.description) else
        if e.name = "vector" then
                writeString ("vector error: " ++ e.description)
        else raise e
in
begin
        let index ← a + 3
        result := vec (index)
end
```

When any exception is raised, either explicitly by a program or by the run-time system, control is transferred to the inner-most exception handler associated with the clause in which the exception occurs. In the example above this handler is the outer **if** clause. The identifier *e* is introduced to denote the exception instance (of type *Exception*); its fields may be accessed to determine the nature of the exception. The result of the exception handling clause, if any, is substituted as the result of the clause in which the exception was raised, and control flow resumes following that clause. In the example, an error message is written out if the name of the exception is "arithmetical" or "vector", otherwise the exception is raised again.

If an exception is raised at a point in a program without a corresponding exception handling clause, ProcessBase searches the dynamic procedure call chain for a handling clause. Thus if a non-handled exception is raised within a procedure body, the exception is handled by the handling clause corresponding to the point of the procedure call, if any. If no handling clause is found anywhere in the call chain, the default exception handler is used. This is illustrated in the following example:

```
let A ← fun (x : int) → int
begin
        if x < 0 do raise Exception ("negative", "fun A requires positive arg")
        x + 2
end

let B ← fun (y : int) → int
        handle exception e1 using
                if e1.name = "negative"  then { writeString ("negative arg "); -1 }
                                         else { raise e1; 0 }
        in A (y - 3)

let C ← fun () → int
        handle exception e2 using {writeString ("exception occurred"); 0 }
        in B (2)

writeInt (C ())
```

The flow of control in this example is as follows. Function *C* is called; its body contains a call to *B* with a corresponding exception handling clause. During the execution of *B* a call to *A* is made, with the argument -1, leading to the raising of an exception with the name "negative" within the body of *A*. Since there is no corresponding exception handling clause at that point, the handling clause corresponding to the point of the call of *A* (in the body of *B*) is executed. This writes out a message and returns the value -1, which in turn forms the return value of *B*. Thus from the point of view of *C*, the call to *B* completes normally, and the value

-1 is returned. If some other exception were to be raised during the execution of *A* or *B*, the handling clause in the body of *C* would be executed and the value 0 returned.

Since procedures are first class values, common exception handling behaviour may be defined within a procedure and used repeatedly, as illustrated below:

```
let voidHandler ← fun (e : Exception)
begin
        writeString ("The exception " ++  e.name ++ " occurred.'n")
        writeString ("Description: " ++ e.description ++ "'n")
        abort ()
end

handle exception e using voidHandler (e)
in x := A (31)

handle exception e using voidHandler (e)
in y := B (4)
```

12.2 Interrupts

ProcessBase interrupts allow communication between the ProcessBase implementation and the ProcessBase language. An event occurring at the implementation level may cause the immediate invocation of a user-defined interrupt handler, written in ProcessBase, with the result returned to the implementation level. From the ProcessBase level this can be thought of as an unexpected procedure call.

Interrupts are similar to standard exceptions in that user-defined handler code may be executed in response to a condition arising at the implementation level. The significant difference is that exception handlers are invoked in response to situations that must be dealt with: exceptions cannot be ignored, and program flow of control is always altered when an exception is raised. In contrast, an interrupt handler may choose to do nothing, in which case the interrupt has no net effect other than the delay involved in invoking the handler. After execution of an exception handler, program flow resumes following the definition of the handler, whereas after execution of an interrupt handler, execution resumes from the point at which the interrupt occurred.

Another difference is that arbitrary new exceptions may be defined and raised by user programs. To accommodate this, exception handlers are generic, in that a single handler handles all kinds of exception occurring in the corresponding clause. In contrast, for any given implementation of ProcessBase the interrupts that may be raised are fixed. Each interrupt handler is specific to one of these statically known interrupts. This design decision, rather than supporting generic handlers, was made on pragmatic grounds: interrupt handling is more likely to be time-critical than exception handling, hence the motivation to avoid the per-interrupt overhead of resolving which interrupt occurred.

The interrupts defined by a particular ProcessBase implementation are specified in the ProcessBase Standard Library [MBG+99c], represented as instances of **interrupt** types. For example, in the following the identifier *clock* represents a hypothetical interrupt that takes a single parameter of type **int** and returns no result:

```
clock : interrupt (int)      ! The type of clock is interrupt (int).
```

An interrupt handler for a particular clause may be introduced with the reserved words **handle interrupt**. The syntax is:

clause ::= **handle interrupt** *interrupt_identifier* **using** *proc_literal* **in** *clause*

$$\frac{T_1,\ldots,T_n \in \text{Type} \quad \tau,\pi \vdash i : \textbf{interrupt}(T_1,\ldots,T_n) \to S \quad \tau,\pi \vdash h : \textbf{fun}(T_1,\ldots,T_n) \to S \quad \tau,\pi \vdash e : T}{\tau,\pi \vdash \textbf{handle interrupt}\ i\ \textbf{using}\ h\ \textbf{in}\ e : T}$$

[handleInterrupt]

The production *interrupt_identifier* in the above signifies that an interrupt identifier defined in the ProcessBase Standard Library must be used. Thus a user-defined alias cannot be used in this context. This restriction is imposed by implementation considerations that require the interrupt being handled to be known statically.

If the specified interrupt occurs during execution of the clause, the handler procedure is called, and control then returns to the point at which the interrupt occurred. If an interrupt occurs at a point where no matching handler is in scope, the dynamic call chain is searched. The first matching handler to be found is called, and again control returns to the point at which the interrupt occurred. Finally, if no handler is found the interrupt is discarded.

There is no analogue to the **raise** clause for interrupts, since interrupts are raised solely by the ProcessBase implementation level.

In the following example the procedure *timedA* uses a handler for the *clockTick* interrupt to accumulate a count of the number of clock ticks that occur during a call of *A*. It is assumed that *clockTick* is raised at regular intervals.

```
let A ← fun ()
begin
        ! Do something useful...
end

let timedA ← fun () → int
begin
        let ticks ← loc (0)

        ! clockTick is an interrupt of type interrupt ().

        handle interrupt clockTick using fun () ; ticks := 'ticks + 1
        in A ()

        'ticks
end

…
let noTicks ← timedA ()
writeInt (noTicks)
```

Here any *clockTick* interrupts occurring before the call to *A* within *timedA* are ignored. From the point of the call to *A*, until *A* returns, each *clockTick* interrupt results in a call to the anonymous handler procedure, incrementing the variable *ticks.* After the return from *A*, further interrupts are again ignored.

There is no built-in interrupt masking mechanism. If a further interrupt is raised during execution of a handler procedure, a further call is made to the corresponding handler for the new interrupt. Handler calls may be arbitrarily nested in this way.

The *interruptOp* down-call allows simple policy to be passed down to the implementation level for specific interrupts. See section 12.3 and the ProcessBase Library Manual [MBG+99c] for details. One possible use is to temporarily disable and re-enable the raising of specific interrupts.

12.3 Down-calls

Corresponding to the up-calls from implementation level to ProcessBase provided by interrupts, ProcessBase also provides down-calls from the language to the implementation level. This is achieved by allowing ProcessBase programs to invoke specific lower level instructions (op-codes) directly.

The op-codes defined by a particular ProcessBase implementation are specified in the ProcessBase Standard Library [MBG+99c], represented as instances of **opcode** types. For example, in the following the identifier *op1* represents a hypothetical op-code that takes two op-code parameters (described below) plus a single stack parameter of type **real**, and returns a result of type **string**:

> op1 : **opcode** [**int**, **int**] (**real**) → **string**
> ! The type of *op1* is ***opcode [int, int] (real) → string***

Two different kinds of parameters to a down-call are distinguished:

- The *op-code parameters*, appearing in square brackets in the op-code type, are those that would normally follow the op-code in the code stream. Whatever their expected size, these parameters are always typed as integers.
- The *stack parameters* are assumed by the instruction implementation to have been pushed onto the stack(s) in the order that they appear between round brackets in the op-code type. Thus the last stack parameter is at the top of the appropriate stack.

The op-code result, if any, is left on the appropriate stack at the end of the instruction, forming the result of the **downcall** clause. Down-calls are written using the following syntax:

clause ::= **downcall** *opcode_identifier* ⟦[*int_literal_list*]⟧ ⟪[*clause_list*]⟫

$$\frac{\forall i \in \{1...n\}\ (\tau,\pi_1 :: < p_i, R_i > :: \pi_2 \ \mathsf{h}\ p_i : \textbf{int})\quad \forall j \in \{1...m\}\ (\tau,\pi\ \mathsf{h}\ e_j : T_j)\quad \tau,\pi\ \mathsf{h}\ op : \textbf{opcode}[R_1,\ldots,R_n](T_1,\ldots,T_m) \to S}{\tau,\ \pi\ \mathsf{h}\ \textbf{downcall}\ op[p_1,\ \ldots,\ p_n]\ \ (e_1,\ \ldots,\ e_m) : S}$$

[downcall]

The production *opcode_identifier* in the above signifies that an op-code identifier defined in the ProcessBase Standard Library must be used. Thus a user-defined alias cannot be used in this context. This restriction is imposed by implementation considerations that require the op-code being down-called to be known statically.

The op-code parameter values are enclosed in square brackets and the stack parameter values in round brackets. The example below shows a down-call to the op-code *op1*. Although all parameters are manifest in this example, in general the stack parameters may be computed dynamically. The stack parameter values are pushed onto the appropriate stacks before the op-code is executed.

```
let aString ← downcall op1[23, 48716](3.1)
```

The **downcall** operation is type-safe for some op-codes and unsafe for others. A set of safe op-codes is defined in the ProcessBase Standard Library; any program that restricts its down-calls to these op-codes is guaranteed to be type-safe. For some applications this may be too restrictive; down-calls may also be made to op-codes defined in other libraries, with the possibility that incorrect use will contravene type-safety.

13 References

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365.

[KCC+92] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed), ISBN 3-540-19800-8 (1992) pp 86-106.

[Kir92] Kirby, G.N.C. “Persistent Programming with Strongly Typed Linguistic Reflection”. In Proc. 25th International Conference on Systems Sciences, Hawaii (1992) pp 820-831, Technical Report ESPRIT BRA Project 3070 FIDE FIDE/91/32.

[MBC+96] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “Napier88 Reference Manual (Release 2.2.1)”. University of St Andrews (1996).

[MBG+99c] Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G.N.C., Mayes, K., Munro, D.S. & Warboys, B.C. “ProcessBase Standard Library Reference Manual (Version 1.0.1)”. Universities of St Andrews and Manchester (1999).

[MCC+93] Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Stemple, D. “Mechanisms for Controlling Evolution in Persistent Object Systems”. Journal of Microprocessors and Microprogramming 17, 3 (1993) pp 173-181.

[Mor79] Morrison, R. “On the Development of Algol”. Ph.D. Thesis, University of St Andrews (1979).

[PS88] “PS-algol Reference Manual, 4th edition”. Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).

[Str67] Strachey, C. “Fundamental Concepts in Programming Languages”. Oxford University Press, Oxford (1967).

[Ten77] Tennent, R.D. “Language Design Methods Based on Semantic Principles”. Acta Informatica 8 (1977) pp 97-112.

Appendix I: Context Free Syntax

Session:

sequence ::= *declaration* [; *sequence*] | *clause* [; *sequence*]

declaration ::= *type_decl* | *value_decl*

Type declarations:

type_decl ::= **type** *type_init* | **rec type** *type_init* [& *type_init*]*

type_init ::= *identifier* **is** *type*

Type descriptors:

type ::= **int** | **real** | **bool** | **string** | **any** | *identifier* |
loc ⟦*type*⟧ | *type | **view** ⟦*labelled_type_list*⟧ |
fun(([*type_list*]) [→ *type*]

type_list ::= *type* [, *type*]*
labelled_type_list ::= *identifier_list* : *type* [; *labelled_type_list*]

Object declarations:

value_decl ::= **let** *value_init* |
rec let *rec_value_init* [& *rec_value_init*]*

value_init ::= *identifier* ← *clause*

rec_value_init ::= *identifier* ← *literal*

Clauses:

clause ::= **if** *clause* **do** *clause* |
if *clause* **then** *clause* **else** *clause* |
while *clause* **do** *clause* |
for *identifier* ← *clause* **to** *clause* [**by** *clause*] **do** *clause*|
project *clause* **as** *identifier*
onto *project_list* **default** : *clause* |
try *clause* **handle** *identifier* **using** *clause* |
raise *clause* |
name := *clause* |
E

project_list ::= *type_id* : *clause* ; [*project_list*]

Expressions:

E ::= *E* **or** *E* | *E* **and** *E* | [~] *E* *rel_op E* | *E add_op E* |
E *mult_op E* | *add_op E* |
literal | *vector_constructor* | (*clause*) |
begin *sequence* **end** | {*sequence*} |
E (*clause* | *clause*) |

E (([*clause_list*])) | **view** (*value_init_list*) |
identifier (*clause_list*) |
'*clause* | **loc** (*clause*) |
clause.*identifier* |
any (*clause*) |
upb (*clause*) | lwb (*clause*) |
name

name ::= *identifier* | *expression* (*clause_list*) [(*clause_list*)]*

clause_list ::= *clause* [, *clause_list*]

value_init_list ::= *value_init* [, *value_init*]

vector_constructor ::= **vector** *vector_element_init*

vector_element_init ::= *clause* **to** *clause* **using** *clause* |
@*clause* **of** [[*clause* [, *clause*]*]]

Literals:

literal ::= *int_literal* | *real_literal* | *bool_literal* | *string_literal* | *view_literal* | *proc_literal*

int_literal ::= *[add_op] digit [digit]**

real_literal ::= *int_literal.[digit]*[e int_literal]*

bool_literal ::= **true** | **false**

string_literal ::= "[*char*]*"

view_literal ::= **nil** (*type*)

proc_literal ::= **fun** (([*labelled_type_list*])) [→ *type*]; *clause*

labelled_type_list ::= *identifier_list* : *type* [; *labelled_type_list*]

Miscellaneous and microsyntax:

add_op ::= + | -

mult_op ::= *int_mult_op* | *real_mult_op* | *string_mult_op*

int_mult_op ::= * | **div** | **rem**

real_mult_op ::= * | /

string_mult_op ::= ++

rel_op ::= *eq_op* | *co_op*

eq_op ::= = | ~=

co_op ::= < | <= | > | >=

identifier_list ::= *identifier* [, *identifier_list*]

identifier ::= *letter* [*id_follow*]

id_follow ::= *letter* [*id_follow*] | *digit* [*id_follow*] | _[*id_follow*]

letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z |
A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

char ::= any ASCII character

Appendix II: Typing Rules

Sequence

$$\frac{\tau,\ \pi\ \vdash A_1 : \textbf{void} \quad \text{Decl}_1 \quad \tau + +\Omega_1,\ \pi + +\Psi_1\ \vdash A_2 : T}{\tau,\ \pi\ \vdash A_1\ ;\ A_2 : T} \qquad \text{[seq]}$$

where Decl_1 stands for typeDecl(Ω_1) and idDecl(Ψ_1) and A_1 and A_2 stand for any constructs in the language

Declarations

$$\frac{\tau \quad \vdash T \in \text{Type}}{\tau \vdash \textbf{type}\ t\ \textbf{is}\ T : \textbf{void} \quad \text{typeDecl}(\ (\ <\ t,\ T\ >\ )\ )} \qquad \text{[typeDecl]}$$

$$\frac{\tau \quad \vdash T_1 \in \text{Type} \quad \tau \quad \vdash T_2 \in \text{Type}}{\tau' \vdash \textbf{rec type}\ t_1\ \textbf{is}\ T_1\ \&\ t_2\ \textbf{is}\ T_2 : \textbf{void} \quad \text{typeDecl}(\ (<t_1, T_1>, <t_2, T_2>)\ )}$$

where τ' stands for $\tau_1::<t_1, T_1>::\tau_2::<t_2, T_2>::\tau_3$ [recTypeDecl]

$$\frac{\tau,\ \pi\ \vdash e : T}{\tau,\ \pi\ \vdash \textbf{let}\ x\ \leftarrow\ e : \textbf{void} \quad \text{idDecl}(\ (\ <\ x,\ T\ >\ )\ )} \qquad \text{[valueDecl]}$$

$$\frac{\tau,\ \pi'\ \vdash e_1 : T_1 \qquad \tau,\ \pi'\ \vdash e_2 : T_2}{\tau,\ \pi\ \vdash \textbf{rec let}\ x_1\ \leftarrow\ e_1\ \&\ x_2\ \leftarrow\ e_2 : \textbf{void} \quad \text{Decl}}$$

where π' stands for $\pi_1::<x_1, T_1>::\pi_2::<x_2, T_2>::\pi_3$ and Decl stands for idDecl(($<x_1, T_1>$, $<x_2, T_2>$)) [recValDecl]

Clauses

Conditional

$$\frac{\tau,\ \pi\ \vdash e : \textbf{bool} \quad \tau,\ \pi\ \vdash e_1 : \textbf{void}}{\tau,\ \pi\ \vdash \textbf{if}\ e\ \textbf{do}\ e_1 : \textbf{void}} \qquad \text{[ifDo]}$$

$$\frac{\tau,\ \pi\ \vdash e : \textbf{bool} \quad \tau,\ \pi\ \vdash e_1 : T \quad \tau,\ \pi\ \vdash e_2 : T}{\tau,\ \pi\ \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : T} \qquad \text{[ifThen]}$$

Iteration

$$\frac{\tau,\ \pi\ \vdash e : \textbf{bool} \quad \tau,\ \pi\ \vdash e_1 : \textbf{void}}{\tau,\ \pi\ \vdash \textbf{while}\ e\ \textbf{do}\ e_1 : \textbf{void}} \qquad \text{[while]}$$

$$\frac{\tau,\ \pi_1::<i, \textbf{int}>::\pi_2\ \vdash e : \textbf{void} \quad \tau,\ \pi\ \vdash e_1 : \textbf{int} \quad \tau,\ \pi\ \vdash e_2 : \textbf{int} \quad \tau,\ \pi\ \vdash e_3 : \textbf{int}}{\tau,\ \pi\ \vdash \textbf{for}\ i\ \leftarrow\ e_1\ \textbf{to}\ e_2\ \textbf{by}\ e_3\ \textbf{do}\ e : \textbf{void}}$$

[for]

Projection

$$\frac{\tau, \pi \vdash e : \textbf{any} \quad T_1, \ldots, T_n \in \text{Type} \quad \forall i \in \{1...n\}\ (\tau, \pi' \vdash e_i : T)}{\tau, \pi \vdash \textbf{project}\ e\ \textbf{as}\ x\ \textbf{onto}\ \texttt{project\_list}\ ;\ \textbf{default}\ e_{n+1} : T}$$

where π' stands for $\pi_1::< x, T_i >::\pi_2$ and project_list stands for $T_1 : e_1 ; \ldots ; T_n : e_n$

[anyProj]

Exception

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi_1 ::< x, \text{Exception} >:: \pi_2 \vdash e_2 : T}{\tau, \pi \vdash \textbf{try}\ e_1\ \textbf{handle}\ x\ \textbf{using}\ e_2 : T}$$ [handle]

$$\frac{\tau, \pi \vdash e : \text{Exception}}{\tau, \pi \vdash \textbf{raise}\ e : \textbf{void}}$$ [raise]

Assignment

$$\frac{\tau, \pi \vdash e_2 : T \quad \tau, \pi \vdash e_1 : \textbf{loc}[\ T\ ]}{\tau, \pi \vdash e_1\ :=\ e_2 : \textbf{void}}$$ [assign]

Expressions

Booleans

$$\frac{\tau, \pi \vdash e : \textbf{bool}}{\tau, \pi \vdash \sim e : \textbf{bool}}$$ [negation]

$$\frac{\tau, \pi \vdash e_1 : \textbf{bool} \quad \tau, \pi \vdash e_2 : \textbf{bool}}{\tau, \pi \vdash e_1\ \textbf{or}\ e_2 : \textbf{bool}}$$ [or]

$$\frac{\tau, \pi \vdash e_1 : \textbf{bool} \quad \tau, \pi \vdash e_2 : \textbf{bool}}{\tau, \pi \vdash e_1\ \textbf{and}\ e_2 : \textbf{bool}}$$ [and]

Comparison

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1\ =\ e_2 : \textbf{bool}}$$ [equality]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1\ \sim=\ e_2 : \textbf{bool}}$$ [nonEq]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1\ <\ e_2 : \textbf{bool}}$$ [less]

where $T \in \{$ **int**, **real**, **string** $\}$

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 \texttt{ <= } e_2 : \textbf{bool}}$$ [lessEq]

where T ∈ { **int**, **real**, **string** }

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 \texttt{ > } e_2 : \textbf{bool}}$$ [greater]

where T ∈ { **int**, **real**, **string** }

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 \texttt{ >= } e_2 : \textbf{bool}}$$ [greaterEq]

where T ∈ { **int**, **real**, **string** }

Arithmetic Expressions

In the following type rules, the type denoted by T ranges over integers and reals i.e. T ∈ { **int**, **real** }

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash \texttt{+}e : T}$$ [plus]

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash \texttt{-}e : T}$$ [minus]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 \texttt{ + } e_2 : T}$$ [add]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 \texttt{ - } e_2 : T}$$ [subtract]

$$\frac{\tau, \pi \vdash e_1 : T \quad \tau, \pi \vdash e_2 : T}{\tau, \pi \vdash e_1 \texttt{ * } e_2 : T}$$ [times]

$$\frac{\tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int}}{\tau, \pi \vdash e_1 \texttt{ div } e_2 : \textbf{int}}$$ [intDiv]

$$\frac{\tau, \pi \vdash e_1 : \textbf{int} \quad \tau, \pi \vdash e_2 : \textbf{int}}{\tau, \pi \vdash e_1 \texttt{ rem } e_2 : \textbf{int}}$$ [intRem]

$$\frac{\tau, \pi \vdash e_1 : \textbf{real} \quad \tau, \pi \vdash e_2 : \textbf{real}}{\tau, \pi \vdash e_1 \texttt{ / } e_2 : \textbf{real}}$$ [realDiv]

$$\frac{\tau, \pi \vdash e_1 : \textbf{string} \quad \tau, \pi \vdash e_2 : \textbf{string}}{\tau, \pi \vdash e_1 \texttt{ ++ } e_2 : \textbf{string}}$$ [concat]

Literals

$$\frac{n \in \text{Integer}}{n : \textbf{int}} \quad \text{[intLiteral]}$$

$$\frac{r \in \text{Real}}{r : \textbf{real}} \quad \text{[realLiteral]}$$

$$\frac{b \in \text{Boolean}}{b : \textbf{bool}} \quad \text{[boolLiteral]}$$

$$\frac{s \in \text{ASCII character set}}{s : \textbf{string}} \quad \text{[strLiteral]}$$

$$\frac{}{\tau_1 ::< t, \textbf{view}[\, l_1 : T_1; \ldots; l_n : T_n \,] >:: \tau_2,\ \pi \vdash \textbf{nil}(\ t\ ) : \textbf{view}[\, l_1 : T_1; \ldots; l_n : T_n \,]} \quad \text{[viewLiteral]}$$

$$\frac{\tau,\ \pi_1 ::< x_1, T_1 >:: \pi_2 :: \ldots ::< x_n, T_n >:: \pi_{n+1} \vdash e : S}{\tau,\ \pi \vdash \textbf{fun}(\ x_1 : T_1,\ \ldots,\ x_n : T_n\ ) \to S\ ;\ e : \textbf{fun}(\ T_1, \ldots, T_n\ ) \to S} \quad \text{[procLiteral]}$$

Vector Constructors

$$\frac{\tau,\ \pi \vdash e_1 : \textbf{int} \quad \tau,\ \pi \vdash e_2 : \textbf{int} \quad \tau,\ \pi \vdash e : \textbf{fun}(\ \textbf{int}\ ) \to T}{\tau,\ \pi \vdash \textbf{vector}\ e_1\ \textbf{to}\ e_2\ \textbf{using}\ e : *T} \quad \text{[vecValue1]}$$

$$\frac{\tau,\ \pi \vdash e : \textbf{int} \quad \forall i \in \{\, 1...n \,\}\ (\tau,\ \pi \vdash e_i : T)}{\tau,\ \pi \vdash \textbf{vector}\ @\ e\ \textbf{of}\ [\ e_1,\ \ldots,\ e_n\ ] : *T} \quad \text{[vecValue2]}$$

Block

$$\frac{\tau,\ \pi \vdash e : T}{\tau,\ \pi \vdash (\ e\ ) : T} \quad \text{[brackets]}$$

$$\frac{\tau,\ \pi \vdash s : T}{\tau,\ \pi \vdash \textbf{begin}\ s\ \textbf{end} : T} \quad \text{[beginEnd]}$$

$$\frac{\tau,\ \pi \vdash s : T}{\tau,\ \pi \vdash \{\ s\ \} : T} \quad \text{[\{\}]}$$

Substring

$$\frac{\tau,\ \pi \vdash e : \textbf{string} \quad \tau,\ \pi \vdash e_1 : \textbf{int} \quad \tau,\ \pi \vdash e_2 : \textbf{int}}{\tau,\ \pi \vdash e\ (e_1 \mid e_2) : \textbf{string}} \quad \text{[substr]}$$

Procedure Application

$$\frac{\forall i \in \{\, 1...n \,\}\ (\tau,\ \pi \vdash e_i : T_i) \quad \tau,\ \pi \vdash e : \textbf{fun}(\ T_1, \ldots, T_n\ ) \to S}{\tau,\ \pi \vdash e(\ e_1,\ \ldots,\ e_n\ ) : S} \quad \text{[procApp]}$$

Vectors

$$\frac{\tau, \pi \vdash e : *\ldots*T \quad \tau, \pi \vdash e_1 : \textbf{int} \ \ldots \ \tau, \pi \vdash e_n : \textbf{int}}{\tau, \pi \vdash e(e_1, \ldots, e_n) : T}$$ [vecProj]

Views

$$\frac{\forall i \in \{ 1...n \} (\tau, \pi \vdash e_i : T_i)}{\tau, \pi \vdash \textbf{view}(l_1 \leftarrow e_1, \ldots, l_n \leftarrow e_n) : T}$$

where T stands for **view**[$l_1 : T_1, \ldots, l_n : T_n$] [viewValue]

Locations

$$\frac{\tau, \pi \vdash e : \textbf{loc}[T]}{\tau, \pi \vdash 'e : T}$$ [locDeref]

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash \textbf{loc}(e) : \textbf{loc}[T]}$$ [locValue]

View Operations

$$\frac{\tau, \pi \vdash e : \textbf{view}[\ldots; l : T; \ldots]}{\tau, \pi \vdash e.l : T}$$ [viewDeref]

Infinite Union

$$\frac{\tau, \pi \vdash e : T}{\tau, \pi \vdash \textbf{any}(e) : \textbf{any}}$$ [anyInj]

Vector Bounds

$$\frac{\tau, \pi \vdash e : *T}{\tau, \pi \vdash \textbf{lwb}(e) : \textbf{int}}$$ [lwb]

$$\frac{\tau, \pi \vdash e : *T}{\tau, \pi \vdash \textbf{upb}(e) : \textbf{int}}$$ [upb]

Identifiers

$$\frac{}{\tau, \pi_1 :: < x, T > :: \pi_2 \vdash x : T}$$ [id]

Appendix III: Program Layout

Semi-Colons

As a lexical rule in ProcessBase, a semi-colon may be omitted whenever it is used as a separator and it coincides with a newline. This allows many of the semi-colons in a program to be left out. However, to help the compiler deduce where the semi-colons should be, it is a rule that a line may not begin with a binary operator. For example,

```
a *
b
```

is valid but,

```
a
* b
```

is not.

This rule also applies to the invisible operator between a vector or view and its index list and between a procedure and its parameters. For example,

```
let b ← a (1,2)
```

is valid but,

```
let b ← a
        (1)
```

will be misinterpreted since vectors can be assigned.

Comments

Comments may be placed in a program by using the symbol !. Anything between the ! and the end of the line is regarded by the compiler as a comment. For example,

```
a + b          ! add a and b
```

Appendix IV: Reserved Words

and	**any**	**as**		
begin	**bool**	**by**		
default	**div**	**do**		
else	**end**			
false	**for**	**fun**		
handle				
if	**int**	**is**		
let	**lwb**	**loc**		
nil				
of	**onto**	**or**		
PS	**project**			
raise	**real**	**rec**	**rem**	
string				
then	**to**	**true**	**try**	**type**
upb	**using**			
vector	**view**			
while				

Index